Problem 1: Remote System Monitoring using RPC

Xu Teng (xuteng@iastate.edu)

Objective

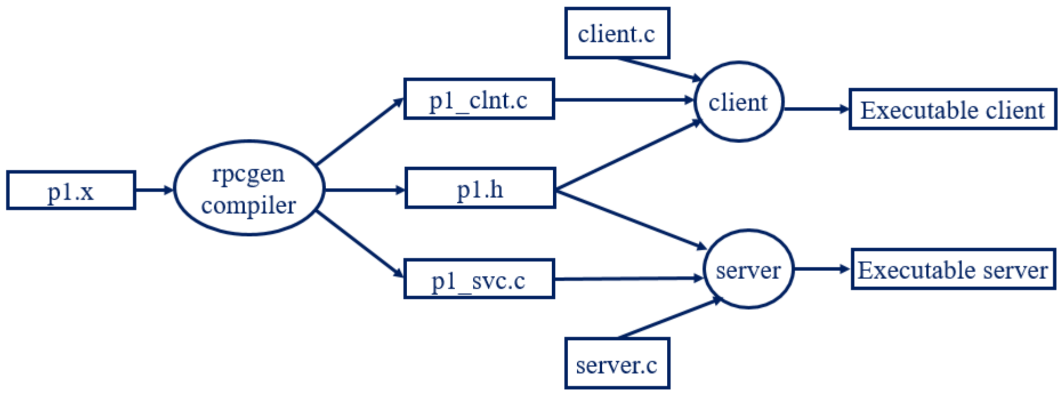
The objective of this work is to implement a remote system monitoring application by using RPC network management system. Feeding the output from monitoring application into an analysis component is helpful for deciding on corrective actions in self-managing distributed systems.

Multiple clients are able to send requests from their own local machine to single/multiple server processes running on a different machine to fetch the system statistics of this server machine, such as

1. Current Date
2. Current Time
3. Combination of Date and Time
4. CPU Usage
5. Memory Usage
6. Swap Usage
7. List of User Names
8. Load Processes per minute

Architecture

The remote monitoring system consists two different types of processes, ***server*** and ***client***. Linux workstation are used as the platform, and whole work is programmed in C language. As mentioned above, RPC is responsible for dealing with (un)marshalling arguments and also handling communication between server and client, which enables clients to communicate with servers by calling procedures in a similar way to the conventional use of procedure calls in high-level languages.

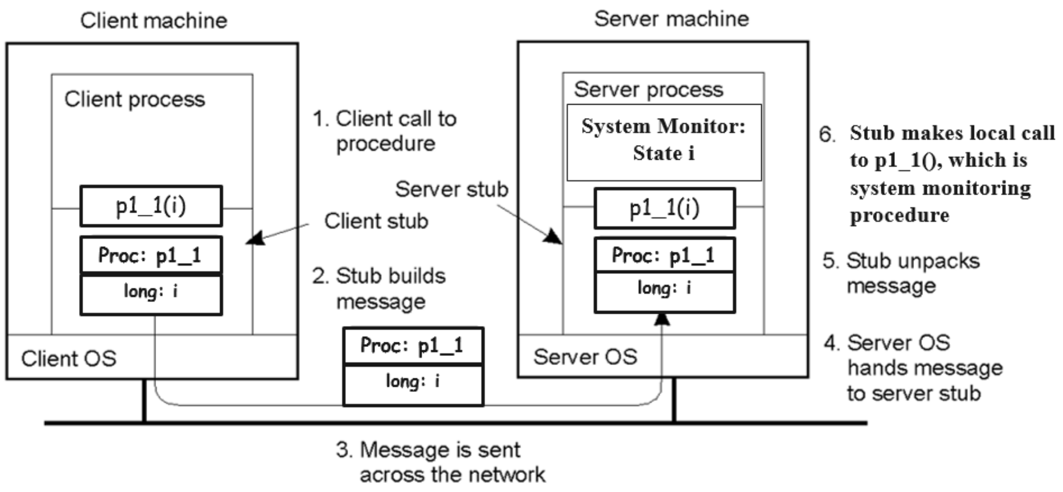


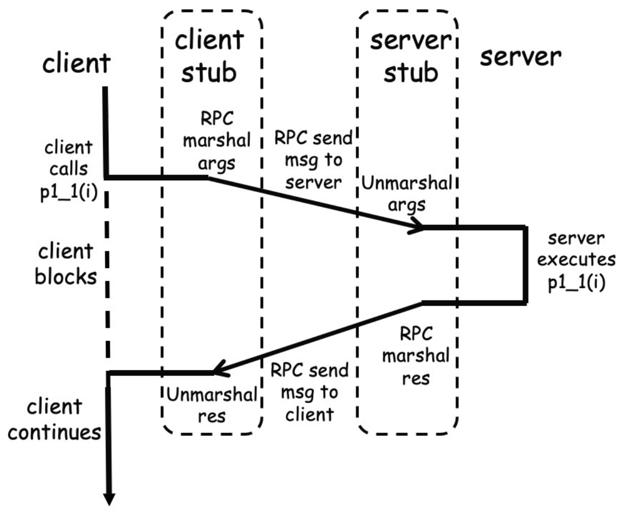
The program generated in the following way:

- **p1.x:** IDL file. Defines service interface between client and server, and specifies the program number, version number and procedure number, which are necessary for creating RPC handle and server/client stub. And also, the data types of the argument sent from client to server and the argument returned from server to client need to be configured in this file, which are long and string in this work.
- **server.c:** After using **rpcgen** complier to read **p1.x** declaration, we will get **p1_svc.c**, which is used for creating server stub, and **p1.h** (header file). Then compiling them with **server.c** by GCC will generate an executable server program. While server is running, it will wait for the calling from server stub and the data types of both input and output are already defined in **p1.x**.
- **client.c:** Similar to server, after compiling **client.c** with **p1_clnt.c** (client stub) and **p1.h** (again, both of which are produced by using **rpcgen** complier to read **p1.x** declaration), GCC will generate an executable client program. Client uses the remote hostname where the server is located (identified by user), the program & version number (pre-defined in **p1.x**) and the type of transport protocol (pre-defined in client.c) to create RPC handle, which is used to invoke RPC call later.

Procedure

After RPC handle on the client side is created successfully, client will fetch the input from keyboard as the argument and call remote monitoring function. RPC client stub, then, marshal the arguments and send it to server stub. On server side, after receiving RPC message, it will invoke the corresponding function with marshaled arguments. The steps of how client calls the remote monitoring procedure on server side are explained below (after step 6, server will marshal the result and send it back to client by using RPC, which is theoretical similar to step 1 ~ step 6). And the second figure below illustrates the whole procedure flow of how RPC works between client and server.





Functions

1. Current Date
   If server stub unmarshals the package and figures out the argument is 1, server process will call **localtime()** function. Because client asks for date information, the server will format the result into {day of the week (%A), month (%B), day (%d), year (%Y)} by using **strftime()** function, and then ask server stub to marshal it and send it back to client.

2. Current Time
   Similar to Date but the only difference here is that server needs to return current timestamp. If server stub unmarshals the package and figures out the argument is 2, after calling **localtime()** function, the server will format the result into 24-hour notation (%T) by using **strftime()** function, and ask server stub to marshal it and send it back to client.

3. Combination of Date and Time
   Similar to Date and Time. The difference between 1, 2 and 3 is the parameter while calling **strftime()** function to format the results of **localtime()** function.

4. CPU Usage
   If server stub unmarshals the package and find the argument is 4, server will open the file ***"/proc/stat"***, and the first line (starting with "cpu") shows the accumulative amount of time that CPU(s) have spent on performing different kinds of work. Apart from "cpu" column, the first 4 columns are helpful for us to calculate CPU usage, which represent the time units spent for normal processes executing in user mode, niced processes executing in user mode, processes running in kernel mode and vacations twiddling thumbs until now (we call it $s_t[1]{\sim}s_t[4]$). Therefore, to get the CPU usage, we have to read that file

twice for some time intervals (in this work, the time interval is set to be 1 second) and calculate the CPU usage as shown below,

$$\frac{\sum_{i=1}^{3}(s_{t+1}[i] - s_t[i])}{\sum_{i=1}^{4}(s_{t+1}[i] - s_t[i])}$$

5. Memory Usage
   If server stub unmarshals the package and find the argument is 5, server will find the memory information by reading the file ***"/proc/meminfo"***, which contains the details about total memory (1st line), available memory (2nd line), buffer memory (3rd line) and cached memory (4th line). And actually, there are lots of other memory information available in this file. For the room of display, this work only presents the first 4 lines.

6. Swap Usage
   If server stub unmarshals the package and find the argument is 6, **sysinfo()** function will be executed on server side and it will return the common information about the state of the system which includes the details about swap usage, i.e., total swap space size (totalswap) and swap space still available (freeswap).

7. List of User Names
   If server stub unmarshals the package and figures out the argument is 7, server process will call **getpwent()** function, which returns a pointer to a structure containing the broken-out fields of a record from the password database. Therefore, each time **getpwent()** is called, a user information will be returned and the pointer will move to the next entry. Eventually, all the user names can be retrieved if calling **getpwent()** multiple times until the pointer reaches end of file (return NULL).

8. Load Processes per minute
   If server stub unmarshals the package and find the argument is 8, server will call **getloadavg()** function, which will return the number of processes in the system run queue averaged over 3 different periods of time (i.e., 1, 5, and 15 minutes). In this work, all 3 averaged load information are fetched by server.

9. Quit
   Control the execution of client progress. If the keyboard input is 9, client will jump out of loop and destroy RPC handle.

Steps to Compile and Run

1. ***ssh*** to Linux workstation (e.g., linux-6.ece.iastate.edu)
2. Save ***p1.x***, ***client.x*** and ***server.c*** into some directory
3. Under the directory, execute:
   a. rpcgen –k p1.x
   b. gcc –o client client.c p1_clnt.c
   c. gcc –o server server.c p1_svc.c
   d. ./server &
   e. ./client localhost (or 127.0.0.1 or IP address shown by running ***hostname –I*** on server machine)
   f. kill server_pid
4. exit